Functional programming in R with functionals

Why a functional style

Functional programming builds a computation out of small, referentially transparent pieces: functions that take input, return output, and do not reach outside themselves to mutate state. In R this style is natural because functions are ordinary values: you can pass them, return them, and store them in lists.

`functionals` is a small toolkit for that style. It gives one consistent family of *functionals* (functions that take a function as an argument), with the same four arguments everywhere:

argument	meaning
`.x` / `.df` / `.l`	the data to iterate over
`.f`	the function to apply
`ncores`	`NULL`/`1` sequential, `>= 2` parallel
`pb`	show a completion-driven progress bar

and, from version 0.6.0, two more that make loops production-safe:

argument	meaning
`.on_error`	`"stop"` (default), `"pass"`, or `"fill"`
`.seed`	reproducible per-task RNG streams

Nothing here depends on `rlang` or the tidyverse; the only import is `parallel`.

The core: `fmap()` and friends

`fmap()` is `lapply()` with the extra arguments above:

```
fmap(1:4, function(x) x^2)
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 4
#>
#> [[3]]
#> [1] 9
#>
#> [[4]]
#> [1] 16
```

The shape-specific mappers save you the `split()` / `as.list()` boilerplate:

```
# one call per data-frame row, row supplied as a named list
fmapr(head(mtcars, 3), function(row) row$mpg / row$cyl)
#> $`1`
#> [1] 3.5
```

```
#>
#> $`2`
#> [1] 3.5
#>
#> $`3`
#> [1] 5.7

# one call per column, with the column name available
fmapc(data.frame(a = 1:3, b = 4:6), function(x, nm) sprintf("%s: %.1f", nm, mean(x)))
#> $a
#> [1] "a: 2.0"
#>
#> $b
#> [1] "b: 5.0"

# one call per group
fmapg(iris, function(d) mean(d$Sepal.Length), by = "Species")
#> $setosa
#> [1] 5.006
#>
#> $versicolor
#> [1] 5.936
#>
#> $virginica
#> [1] 6.588

# one call per aligned tuple (list names are matched to .f's arguments)
fmapn(list(x = 1:3, y = 4:6), function(x, y) x + y)
#> [[1]]
#> [1] 5
#>
#> [[2]]
#> [1] 7
#>
#> [[3]]
#> [1] 9
```

`fwalk()` is the side-effect sibling: it runs `.f` for its effect and returns `.x` invisibly.

```
invisible(fwalk(1:3, function(i) cat("row", i, "\n")))
#> row 1
#> row 2
#> row 3
```

`floop()` is deprecated

`floop()` did nothing that `fmap()` (collect results) and `fwalk()` (side effects only) do not. As of 0.6.0 it warns and forwards to them; use those directly.

Programming *with* functionals

Function factories

A function that returns a function pairs naturally with `fmap()`:

```r
power_of <- function(k) function(x) x^k
powers   <- fmap(1:4, power_of)          # a list of functions
powers[[3]](2)                            # 2^3
#> [1] 8
```

Composition and reduction

`fcompose()` chains single-argument functions right to left; `freduce()` folds a binary function over a sequence, optionally keeping the running accumulation.

```r
standardise <- fcompose(function(x) round(x, 2), scale, as.numeric)
standardise(c(1, 2, 3, 4))
#>       [,1]
#> [1,] -1.16
#> [2,] -0.39
#> [3,]  0.39
#> [4,]  1.16
#> attr(,"scaled:center")
#> [1] 2.5
#> attr(,"scaled:scale")
#> [1] 1.290994

freduce(1:5, `+`)
#> [1] 15
freduce(1:5, `+`, .accumulate = TRUE)
#> [1]  1  3  6 10 15
```

A small pipeline built entirely from functionals, fitting nothing, just transforming:

```r
pipeline <- fcompose(
  function(x) Filter(function(v) v > 0, x),
  function(x) fmap(x, function(v) v - mean(unlist(x)))
)
str(pipeline(list(-2, 1, 4, 9)))
#> List of 2
#>  $ : num 1
#>  $ : num 6
```

Parameter sweeps with `fmapn()`

`fmapn()` iterates over aligned lists, so a grid of settings becomes a single call. `expand.grid()` builds the grid; `fmapn()` walks it.

```r
grid <- expand.grid(n = c(10, 50), sd = c(1, 2))
res  <- fmapn(
  list(n = grid$n, sd = grid$sd),
  function(n, sd) c(n = n, sd = sd, se = sd / sqrt(n))
)
do.call(rbind, res)
#>       n sd        se
#> [1,] 10  1 0.3162278
#> [2,] 50  1 0.1414214
#> [3,] 10  2 0.6324555
#> [4,] 50  2 0.2828427
```

Safe iteration

With `.on_error = "pass"`, one bad element does not abort the run; the failed slot holds a `functionals_error` you can inspect afterwards.

```
out <- fmap(list(1, "oops", 3), function(x) x * 10, .on_error = "pass")
#> Warning: 1 of 3 element failed; returned as <functionals_error> (inspect with
#> conditionMessage()).
vapply(out, function(e) if (inherits(e, "functionals_error")) NA_real_ else e, numeric(1))
#> [1] 10 NA 30
```

`.on_error = "fill"` replaces failures with `.fill` directly. The behaviour is identical whether the run is sequential or parallel.

Reproducible randomness

`.seed` gives each task an independent L'Ecuyer-CMRG stream, so results do not depend on `ncores` or the order tasks finish, and your global RNG state is left untouched.

```
a <- fmap(1:5, function(i) round(rnorm(1), 3), .seed = 42)
b <- fmap(1:5, function(i) round(rnorm(1), 3), .seed = 42, ncores = 1)
identical(a, b)
#> [1] TRUE
```

Parallelism and progress

Set `ncores` to fan work across processes (fork on Unix, PSOCK elsewhere or when you pass your own `cl`). `pb = TRUE` shows a bar that advances as individual tasks complete and reports elapsed time and ETA.

```
slow <- function(x) { Sys.sleep(0.2); x^2 }
fmap(1:20, slow, ncores = 4, pb = TRUE)
```

Objects that cannot survive a fork (external-pointer handles from `torch`, `xgboost`, database connections, ...) must be created *inside* `.f`, not captured from the caller. When that is impossible, run sequentially.

`functionals` and `purrr`

The vocabularies line up almost one to one.

Task	`functionals`	`purrr`	base R
map	`fmap(x, f)`	`map(x, f)`	`lapply(x, f)`
map over N inputs	`fmapn(list(x, y), f)`	`pmap(list(x, y), f)`	`Map(f, x, y)`
map over rows	`fmapr(df, f)`	`pmap(df, f)`	`apply(df, 1, f)`
map over columns	`fmapc(df, f)`	`imap(df, f)`	`Map(f, df, names(df))`
grouped map	`fmapg(df, f, by = "g")`	`map(split(df, df$g), f)`	`lapply(split(df, df$g), f)`
walk	`fwalk(x, f)`	`walk(x, f)`	`for (i in x) f(i)`
reduce	`freduce(x, `+`)`	`reduce(x, `+`)`	`Reduce(`+`, x)`
compose	`fcompose(f, g)`	`compose(f, g)`	`function(x) f(g(x))`
repeat / simulate	`frepeat(times = n, expr = f)`	`map(seq_len(n), ~ f())`	`replicate(n, f())`
parallel + progress	`fmap(x, f, ncores = 4, pb = TRUE)`	`furrr::future_map()` + `progressr`	`parLapply(cl, x, f)`

Differences that matter:

- **Dependencies.** `functionals` imports only `parallel`. `purrr` pulls in `rlang`, `cli`, `vctrs`, and more.
- **Anonymous functions.** `functionals` takes plain closures only, with no `~ .x + 1` formula shorthand. R's native `\(x) x + 1` covers most of that gap.
- **Parallelism is built in.** In `purrr` you switch to `furrr` and set a `future` plan; here it is the `ncores` argument.
- **Type-stable output.** `purrr` has `map_dbl()`, `map_chr()`, ... `functionals` always returns a list; simplify yourself with `unlist()`, `vapply()`, or `do.call(rbind, .)`.

Performance versus `purrr`

The script `system.file("benchmarks/bench-purrr.R", package = "functionals")` compares the engines **single-threaded** (`ncores = NULL`, `pb = FALSE`, `mc.cores = 1`), with an identical closure for every engine, across problem sizes and two payload weights. The results below are precomputed; re-run the script to reproduce them on your machine.

```
#> [1] n       payload
#> <0 rows> (or 0-length row.names)
```

Timings in milliseconds (median). Linux, R version 4.5.1 (2025-06-13); functionals 0.6.0, purrr 1.2.2.

The takeaway is deliberately unglamorous: with one thread and a cheap `.f`, all three are within a small constant factor: `fmap()` is `lapply()` plus an argument check, and `purrr::map()` is also compiled C. Raw sequential speed is not why you reach for `functionals`. The reasons are the *uniform* interface, built-in `ncores` parallelism, the completion-driven progress bar, and the `.on_error` / `.seed` controls that the next vignette puts to work in a modeling pipeline.